<u>SAMPLE PROBLEM</u>

<u>0. Petascale</u>
Program File: petascale.java, petscale.py, petscale.c
Input File: None

TACC’s first cluster-based HPC system was **RANGER**, operating at a peak capacity of 540 TeraFLOPS. That’s 540 trillion floating operations per second. A PetaFLOP is 1000 TeraFLOPs - the current scale of computing. Our newest system, **FRONTERA** will operate at approximately 40 PetaFLOPS. **RANGER**, while still deployed, filled up half of the data center. **FRONTERA** will take up less than a quarter of the space.

Your program must output how many times faster **FRONTERA** is than **RANGER**, rounded down to the nearest whole number.

INPUT *None*

OUTPUT

```
74
```

TACC Invitational 2019 Hands-On Programming Packet

I. General Notes

- There are 12 problems. All problems may be done in any order and are worth 60 points
- There is no extraneous input. All input is exactly as specified in the problem. Unless otherwise specified, your program should read input to the end of the file
- Your program should not print extraneous output. Follow the form exactly as given in the problem.
- A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Problem Names

	Problem Name	**Program File**	**Input File**
1.	Paginator	paginator.py, .java	paginator.dat
2.	Allocations	allocations.py, .java	allocations.dat
3.	Polynomial	polynomial.py, .java	polynomial.dat
4.	Queens	queens.py, .java	queens.dat
5.	Change Detection	changedetection.py, .java	changedetection.dat
6.	Merge Conflicts	mergeconflicts.py, .java	mergeconflicts.dat
7.	Onboarding	onboarding.py, .java	onboarding.dat
8.	ElasticSearch	elasticsearch.py, .java	elasticsearch.dat
9.	Network Route	networkroute.py, .java	networkroute.dat
10.	Classify Images	classifyimages.py, .java	classifyimages.dat
11.	Oversubscribed	oversubscribed.py, .java	oversubscribed.dat
12.	Smith-Waterman	smithwaterman.py, .java	smithwaterman.dat

1. Paginator
Program File: paginator.py, paginator.java
Input File: paginator.dat

Sometimes, it's necessary to paginate extremely long lists of data. A user can then browse to a certain page and see all of the "neighboring" pages. A paginator user interface element should be able to compute which page numbers to show based off of the current page number being viewed and the number of pages that are available. Assuming that a paginator shows 5 pages, it should show a user's current page number and up to two neighboring pages in either direction if they are available. If there are even more pages in either direction, it should show an ellipsis.

Your program will accept a series of number pairs, where the first number is the currently viewed page and the second number is the number of total pages. It must output a paginator display.

INPUT paginator.dat

```
4 9
2 3
5 7
```

OUTPUT

```
... 2 3 4 5 6 ...
1 2 3
... 3 4 5 6 7
```

2. Allocations

Program File: allocations.py, allocations.java
Input File: allocations.dat

TACC has many different types of resources, including execution systems and storage systems. Users must request an Allocation for any particular system, and include the number of Service Units (SUs) required, based on their expected usage of the system.

The storage systems **CORRAL** and **RANCH** use 1 SU per 1 gigabyte stored. Sometimes, users specify usage in terabytes (1024 gigabytes) and petabytes (1024 terabytes). Execution systems run computational tasks and use SUs according to the following equations:

STAMPEDE2 and **LONESTAR5** SUs = nodes x hours x jobs
HIKARI SUs = cores x nodes x hours x jobs

There is a special system called **WRANGLER**, which is a data-intensive execution system. Users will request both nodes and data storage.

WRANGLER SUs = nodes x hours x jobs + gigabytes

Your program must accept a series of inputs describing a user's system requirements and output the number of SUs they will need. Units will be given in the order of the calculations listed above, and each line will always have all required units.

INPUT allocations.dat

```
STAMPEDE2 200 NODES 3 HOURS 20 JOBS
LONESTAR5 150 NODES 10 HOURS 10 JOBS
HIKARI 60 CORES 10 NODES 4 HOURS 4 JOBS
CORRAL 2 TERABYTES
RANCH 3 PETABYTES
WRANGLER 20 NODES 1 HOURS 1 JOBS 200 GIGABYTES
```

OUTPUT

```
12000
15000
9600
2048
3145728
220
```

3. Polynomial

Program Files: polynomial.py, polynomial.java
Input File: polynomial.dat

Write a program to take an input line representing a polynomial equation in a single variable, X, and a value for X and compute the value of the polynomial for the given value of X. The polynomial will involve only integer coefficients and positive, integer exponents. Additionally, the polynomial will be written in standard form; the only characters used in the polynomial string will be X, +, -, 0, through 9 and ^ for exponentiation, and spaces between terms are allowed but not within a single term.

INPUT polynomial.dat

```
3X^11 -12X^5 +6X^2 -102X +17
X=3
```

OUTPUT

```
4
```

4. Queens
Program Files: queens.py, queens.java
Input Files: queens.dat

Chess is a board game played on an 8x8 square grid with pieces that can move according to certain rules. A piece can capture another piece if it can legally move to the square the piece occupies. The queen is the most powerful piece in the chess: on a given turn, a queen can choose to move either horizontally, vertically, or diagonally, and it can move any number of squares in its chosen direction. For example, if we assume the rows of the chess board are labeled A through H and the columns are labeled 1 through 8, then a queen initially in square A1 can move to any square in row A (i.e., A2, A3, …, A8), any square in column 1 (i.e., B1, C1, D1, …, H1), and any square along the diagonal (i.e., B2, C3, D4, …).

Write a program to take as input a number of queens and the locations of the queens on a chess board and determine if any queen can capture any other queen in one move. The first line of the input will be an integer, N > 0, representing the number of queens and there will be N subsequent lines of input representing the locations of the queens.

Your program should output “True” if any queen can capture any other queen in a single move and “False” otherwise.

INPUT queens.dat

```
3
B3
C7
H2
```

OUTPUT

```
TRUE
```

5. Change Detection

Program File: changedetection.py, changedetection.java
Input File: changedetection.dat

A popular front-end framework for web development is Angular. It hooks into a web page and allows you to define your own components. In addition, when component data changes in the web browser, automated change detection allows the page to update itself. For efficiency, change detection only occurs on child elements.

Your program must accept, as input, a web document and some change events. The web document will always have one root element HTML /HTML. The document will be followed by a list of CHANGE events, listing the element that changed. Your program must output each immediate child element that changed.

INPUT changedetection.dat

```
HTML
NAME
/NAME
INFO
ADDRESS
STREET
/STREET
CITY
/CITY
/ADDRESS
EMAIL
USERNAME
/USERNAME
DOMAIN
/DOMAIN
/EMAIL
/INFO
/HTML
CHANGE ADDRESS
CHANGE EMAIL
CHANGE INFO
CHANGE HTML
```

OUTPUT

```
STREET CITY
USERNAME DOMAIN
ADDRESS EMAIL
NAME INFO
```

6. Merge Conflicts
Program File: mergeconflicts.py, mergeconflicts.java
Input File: mergeconflicts.dat

The Git source control software is commonly used when many people collaborate on a software project. Files and a history of changes are stored in “repositories”, of which there is one “master” branch. Groups of changes to several files (as well as new files) are called “commits”. A group of commits is called a “branch”. These branches can be merged into the master.

Sometimes, when branches are merged, they may have files where changes were committed that conflict. Specifically, if an older commit is applied on top of a newer commit, a “merge conflict” arises.

Your program will accept a series of git commands that occur chronologically. The commands are either COMMIT *Branch_Name File1, FIle2, File3…* or MERGE *Branch_Name*. It will output which files were “auto merged” (as in, there are no chronological conflicts) and which ones generated “merge conflicts”. If there are merge conflicts, the file is not updated.

INPUT mergeconflicts.dat

```
COMMIT MASTER FILE1 FILE2
COMMIT BRANCH_A FILE1 FILE2 FILE3
COMMIT BRANCH_B FILE2 FILE3
MERGE BRANCH_B
MERGE BRANCH_A
```

OUTPUT

```
AUTOMERGE FILE2 FILE3
AUTOMERGE FILE1 CONFLICT FILE2 FILE3
```

7. Onboarding

Program File: onboarding.py, onboarding.java
Input File: onboarding.dat

When users sign up for accounts at TACC, they must go through a series of onboarding steps. Once every step has been marked COMPLETE, a user is considered SETUP_COMPLETE. If a step is not COMPLETE, it is considered to have a failure state.

Your program must accept a sequence of onboarding steps and usernames with events and output each user in alphabetical order, listing either the earliest sequence failure event and state, or “SETUP_COMPLETE”. If a step event for a user has not occured yet, list it as INCOMPLETE.

INPUT onboarding.dat

```
STEPS APPROVAL MFA ALLOCATION
JCHUAH MFA NOTPAIRED
JSTUBBS APPROVAL COMPLETE
JMEIRING MFA COMPLETE
JSTUBBS ALLOCATION COMPLETE
JCHUAH APPROVAL COMPLETE
JCHUAH ALLOCATION COMPLETE
JMEIRING ALLOCATION COMPLETE
JSTUBBS MFA COMPLETE
```

OUTPUT

```
JCHUAH MFA NOTPAIRED
JMEIRING APPROVAL INCOMPLETE
JSTUBBS SETUP_COMPLETE
```

8. ElasticSearch

Program File: elasticsearch.py, elasticsearch.java
Input File: elasticsearch.dat

ElasticSearch (ES) is a technology that performs extremely fast search on content by using multiple Shards. Let's say you want to make a file searchable using ES. ES indexes the file for search terms, and then splits the index data across multiple Shards. When a search is requested, the Shards return the score of each indexed file and the combination score is used to generate search results.

Your program must accept, as input, a series of ES Shard indexes, followed by a search.
It must output each file that appears in every shard in order from greatest search score to least search score. The score is based on the order in which a word occurred in a file's index, with earlier occurrences being scored higher. For example, the word DNA would give FILE_A a score of 2 from the Shard 0, and FILE_B a score of 3 from Shard 1. FILE_C would receive a score of 0 for this search term.

INPUT elasticsearch.dat

```
SHARD
FILE_A PROTEIN DNA RNA
FILE_B ENZYME STRUCTURE RNA
FILE_C DNA CIRCUITS RULE30
SHARD
FILE_A SEQUENCING ENZYME YEAST
FILE_B DNA FERMENTATION METABOLOMICS
FILE_C CRISPR RIBOSOME CELL
SHARD
FILE_A PROTEOMICS METABOLOMICS VARIANTS
FILE_B WILDTYPE YEAST GENOTYPE
FILE_C SEQUENCING TRANSCRIPTASE GATES
SEARCH EFFECTS OF RNA TRANSCRIPTASE IN CRISPR ON YEAST
```

OUTPUT

```
FILE_C
FILE_B
FILE_A
```

9. Network Route
Program Files: networkroute.py, networkroute.java
Input File: networkroute.dat

Typically, the graph associated with a computer network is not complete; that is, a given computer is not directly connected to all the other computers on a network, only a subset of them. Since a given computer can only send a message packet to another computer to which it is directly connected, a single packet may pass through multiple computers before it reaches its final destination.

Your program must accept a series of input lines describing how 5 computers labeled A through E are connected in a network as well as a “Message” line with a source and a destination computer to route a message. You program should compute the least number of computers, also known as “hops”, through which the message must be passed. The destination computer counts as a hop, so in the case where the two computers are directly connected, your program should return 1.

For input lines that begin with “Computer”, the first letter is the label of the current computer and the subsequent letters are the computers that computer can send a message to. For instance, in the example input below, Computer A can send a message to Computers B, D, and E, (line 1) Computer B can send a message to computer C only (line 2), Computer C can send a message to Computers D and E (line 3), and so on.

INPUT networkroute.dat

```
COMPUTER A B D E
COMPUTER B C
COMPUTER C D E
COMPUTER D A B E
COMPUTER E C B
MESSAGE C B
```

OUTPUT

```
3
```

10. Classify Images
Program Files: classifyimages.py, classifyimages.java
Input File: classifyimages.dat

You have a brand new machine learning algorithm that can classify an image of a tumor as either cancerous or benign, and you have millions of images you need to classify for the Dell Medical School using your algorithm. You received an allocation of 50,000 Service Units (SUs) at TACC to run jobs on different machines to classify the images. Some of the machines run your algorithm more efficiently than others and can therefore classify more images in a single job. Additionally, a single job costs a different number of SUs on each system, and there is a maximum of 3 jobs allowed on a given system at a time.

Your program must accept a series of input lines with each line describing the name of an available system, the number of images that can be classified in a single job on that system, and the number of SUs a job will cost for that system. Your program should output the maximum number of images that can be classified by first determining how many jobs to run on each system. Assume you can submit no more than 3 jobs to a single system.

INPUT classifyimages.dat

```
LONESTAR5 25000 12500
STAMPEDE2 50000 18000
WRANGLER 35000 14000
MAVERICK 20000 25000
HIKARI 8000 10000
```

OUTPUT

```
25000
```

11. Oversubscribed

Program Files: oversubscribed.py, oversubscribed.java
Input File: oversubscribed.dat

Different systems at TACC have different numbers of machines or “nodes” associated with them. When a user submits a job to a system, they ask the scheduler for a certain number of nodes for a certain amount of time. The goal with any HPC system is to keep as many nodes in use as possible at all times. Since a given job might be cancelled by the user, schedulers will sometimes “oversubscribe” the number of nodes available in the system in an effort to keep as many nodes in use as possible; that is to say, the scheduler will scheduler more jobs than it has nodes available at a given time, expecting one or more jobs to be cancelled before the start time.

Write a program to accept a set of jobs and determine which (if any) systems have been oversubscribed. A system is considered “oversubscribed” if at any given moment in time, the jobs scheduled to run on the system total more nodes than the total number of nodes in the system. Assume each job will be scheduled on one of the three systems below, with the total number of nodes listed:

Stampede2: 4,200 nodes
LoneStar5: 1,252 nodes
Hikari: 432 nodes

The first line of input will be an integer representing the number of jobs. Each subsequent line of input represents one job. Each input job line is a comma separated string in the following format: name of system job is scheduled to run on, start time of job, total run time (in hours) of job, number of nodes for job. Assume the start time is in military time (00:01 - 23:59) and jobs start on the hour (e.g., 8:00, 9:00, etc.), and that all jobs are scheduled to run on the same day. Assume the run time of each job is a positive integer number of hours.

INPUT oversubscribed.dat

```
9
STAMPEDE2 1:00 8 3000
STAMPEDE2 14:00 9 4000
LONESTAR5 2:00 4 1000
STAMPEDE2 18:00 6 4000
HIKARI 9:00 6 400
LONESTAR5 8:00 9 1000
LONESTAR5 18:00 4 1000
HIKARI 15:00 6 400
HIKARI 20:00 1 50
```

OUTPUT

```
STAMPEDE2
HIKARI
```

12. Smith-Waterman

Program File: smithwaterman.py, smithwaterman.java
Input File: smithwaterman.dat

The Smith-Waterman Algorithm is used to compare genetic amino acid sequences (GTTACTAC...) reasonably quickly. The challenge with comparing sequences is that due to experimental conditions such as errors or mutation sequences may have slight variations such as missing amino acids or minor mismatches, but otherwise form a very good match.

To accomplish this, the Smith-Waterman algorithm generates a matrix of values. Starting with the largest value, it begins "walking back" toward the upper left corner of the matrix, following the path to the highest adjacent value.

- Any time it moves diagonally to the upper right, it stores the corresponding row and column amino acids as a match. (If they don't actually match, it's okay. What harm is there in a little mutation?)
- Any time it moves to directly up, it records that there is a missing amino acid in the sequence represented by the columns
- Any time it moves to the left, it records that there is a missing amino acid in the sequence represented by the rows.

For example, the Smith-Waterman matrix:

X	A	C	G	C	A	T	A	T	G
G	0	0	**1**	0	0	0	0	0	1
C	0	1	0	**2**	0	0	0	0	0
G	0	0	2	0	**1**	0	0	0	1
T	0	0	0	1	0	**2**	0	1	0
A	1	0	0	0	2	0	**3**	1	0
T	0	0	0	0	0	3	1	**4**	2

Would show that the most similar sequence would be:

```
GCGTAT
GCATAT
```

Your program must accept a series of Smith-Waterman matrices and output the matching amino acid sequence pairs, with the row sequence first and the column sequence second.

INPUT smithwaterman.dat

```
5 ROWS 6 COLUMNS
X A  T  G  A  T  C
T 0  3  1  0  3  1
G 0  1  6  4  2  2
A 3  1  4  9  7  5
T 1  6  4  7  12 10
C 0  4  5  5  10 15
5 ROWS 6 COLUMNS
X G  A  T  T  A  C
G 3  1  0  0  0  0
C 1  2  0  0  0  3
T 0  0  5  3  1  1
T 0  0  3  8  6  4
A 0  3  1  6  11 9
8 ROWS 7 COLUMNS
X G  T  A  T  C  G  C
G 3  1  0  0  0  3  1
C 1  2  0  0  3  1  6
T 0  4  2  3  1  2  4
A 0  2  7  5  3  1  2
T 0  3  5  10 8  6  4
C 0  1  3  8  13 11 9
G 3  1  1  6  11 16 14
A 1  2  4  4  9  14 15
```

OUTPUT

```
ROW TGATC
COLUMN TGATC
ROW GCTTA
COLUMN GATTA
ROW GCTATCG
COLUMN G-TATCG
```